Preventing Bug Reopening With Effective Software Verification

Rodrigo Souza Christina Chavez Roberto Bittencourt

July 2, 2013

Abstract

Many companies perform software verification on bug fixes to determine if they are appropriate. Verification can spot missed corner cases and other problems, thus preventing the shipping of buggy code. Software verification processes vary in dimensions such as time (when is verification performed?), people (is there a specific role responsible for verifications?), and technique (code review, ad hoc testing, and so on). What is the optimal software verification process so that it catches most problems with bug fixes before they are released to the public? Answering this question would be important to software companies, that would release higher quality products, as well as to the research community.

1 Introduction

Fixing bugs is an important software development task, often supported by bug tracking systems [6]. In a typical scenario, a bug report is created, then the bug is fixed, and finally the report is closed. Sometimes, however, someone discovers that the fix was incomplete or innapropriate. In this case, the bug report is reopened so it can be further investigated.

Reopened bugs reduce the software development productivity, because they lead to rework and take twice as much time to fix, on average [7]. Also, they involve the participation of 20 to 60% more developers [5].

Even worse is when a bug receives an incomplete fix but developers only notice it after the software has been released. In this case, the bug is reopened only after the release, already too late, and the software quality, as perceived by users, is impaired.

To avoid post-release bug reopenings, in some projects the bug fixes are verified. Verifying a bug fix means that someone independently checks if it is appropriate and complete. If it is not, the bug report is reopened right away, so it can be properly fixed before the next release.

Of course, verification is not perfect, and a bug can be reopened even after it has been considered appropriate during its verification. The verification, in this case, is considered unsuccessful.

Why some verifications are successful and others are not? The complexity of fixing a particular bug surely plays an important role, but it cannot be controlled. If we want to avoid post-release bug reopening, we should control how verification is performed.

The verification process varies in many dimensions, e.g., is it performed by a specialized team?, is it concentrated in specific time periods?, does it involve automated testing, code review etc.? There is little empirical evidence, however, on how these and other dimensions affect the success rate of verifications.

2 Problem

We propose to investigate which dimensions contribute to successful verifications, so post-release bug reopenings are minimized. We plan to do so by analyzing bug reports, specially by comparing bugs that were reopened after verification and bugs that were not. In the end, we expect to find empirically supported guidelines for software verification processes.

3 Background

The term reopening comes from bug tracking systems such as Bugzilla[1]. In those systems, bugs that are not resolved are considered open. Thus, if a bug is resolved and, later on, it is found that the resolution is not appropriate, the bug is reopened.

Usually a developer resolves a bug by fixing it, but this is not the only possible resolution. During bug triage, a bug report can be resolved by tagging it as invalid (i.e., it is not a bug), incomplete (there is not enough information to reproduce it), "won't fix" (the bug is not worth fixing), among other resolutions.

Therefore, in the broad sense, a bug is considered reopened if its triage was not definitive, e.g., if the report was incomplete at first, but then new information arrived that helped developers reproduce it. In this proposal, we use the term reopening in a stricter sense, meaning the bug was considered resolved by applying a fix, but the fix was later considered innapropriate.

4 Related Work

Despite its importance, only recently researchers have started to study bug reopening. Shihab and colleagues [8, 7] developed a decision tree model, based on features about the bug report, the bug fix, and human factors, to predict which bugs would be reopened (in the broad sense). They found that certain components are more prone to reopening (such as those dealing with concurrency).

Zimmermann and colleagues [12] asked Microsoft engineers what are common causes for bug reopening (in the broad sense). Responses included the difficulty to reproduce a bug, the misunderstanding of root causes, the lack of information in the initial report, the increase of the bug priority, incomplete fixes, and code integration problems.

They also have built classification models to predict reopened bugs. They found that bugs found by users are more likely to be reopened than those found by code review or static analysis, supposedly because those reported by users are harder to and reproduce and tend

[1] `http://www.bugzilla.org/`

to be more complex. According to the authors, other factors that favor reopening include bug severity and geographical distribution of developers participating in the bug. We argue that most of these factors are not controllable, and therefore cannot be used to prevent the reopening in the first place.

Some works deal specifically with bugs that were reopened after receiving a fix. Jongyindee and colleagues [3] found that bugs are less likely to reopen when fixed by more active developers. Almossawi [1] concluded that bugs located in code with high cyclomatic complexity are more likely to be reopened after fix.

Both Shihab's [8, 7] and Zimmermann's [12] works investigate reopening in a broad sense. In our opinion, the reopening of fixed bugs and the reopening of bugs that were just triaged are different phenomena, with distinct causes and prevention approaches. In this proposal, we aim to investigate the reopening of fixed bugs.

Also, most findings are of limited utility for those trying to prevent bug reopening. For example, although software component is a strong predictor of bug reopening [8], usually it is impossible to avoid the appearance of new bugs on bug-prone components. In this proposal, we are interested in preventing bug reopening by means of software verification.

5 Intellectual Merits

Current thinking: predicting reopened bugs. But what do to if a bug is likely to be reopened? Concentrate verification efforts?

Originality: my proposal is different because it focus on impacts and prevention of reopened bugs that were considered fixed. (Prevention = verification.)

Impacts: discover how verifications processes can be optimized in order to avoid bug reopening. We expect that such discovery could shed some light on how to tune the verification process to prevent that new features introduce bugs.

6 Why this research problem is important to the community

7 Data and Methods

Based on Figure 3 of CODEMINE's whitepaper [2], we need access to the following types of data: work item, organization, code review, process information, and, possibly, build and test (if they can be related to work items). The analysis will be centered on work items, and the other types of data will provide context to allow more refined inferences (e.g., developer role at Microsoft, how bug was discovered [12]). The project would also benefit from interviews or surveys with Microsoft engineers.

We expect to conclude the project in 8 weeks, according to the following schedule:

1. learn about CODEMINE schema, API and tools
2. run a survey about software verification at Microsoft
3. TODO

4. TODO

5. TODO

6. TODO

7. opportunistic interviews [4]

8. TODO

8 People and Publications

One PhD student and two professors are involved in the project. Below, a short bio for each one.

Rodrigo Souza is a computer science PhD student at Federal University of Bahia, Brazil, with master degree in from the Federal University of Campina Grande, Brazil. He has already done some work to support this proposal as part of his thesis, using data from open source projects. He has experience in academia, industry, and open source projects. He gives talks about data analysis with the R programming language and is an active contributor of the Data Analysis Community Portal[2].

Christina Chavez is ...

Roberto Bittencourt is ...

We have published three papers that supports this proposal. The first one was a characterization of the verification process of open source projects using data from bug reports. The results were presented at the 9th Working Conference on Mining Software Repositories (MSR 2012) [9].

This year we presented two papers [10, 11] on the Data Analysis Patterns in Software Engineering workshop, organized by Microsoft Research and held in San Francisco, together with the 35th International Conference on Software Engineering (ICSE 2013). The papers include four patterns that help data scientists clean and transform bug data before it can be analyzed.

References

[1] A. Almossawi. Investigating the architectural drivers of defects in open-source software systems: an empirical study of defects and reopened defects in gnome, 2012.

[2] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy. CODEMINE: Building a software analytics platform for collecting and analyzing engineering process data at Microsoft. Technical Report MSR-TR-2013-7, Microsoft Research, 2013.

[3] A. Jongyindee, M. Ohira, A. Ihara, and K.-i. Matsumoto. Good or bad committers? a case study of committers' cautiousness and the consequences on the bug fixing process in the eclipse project. In *Proceedings of the 2011 Joint Conference of the 21st International*

[2] `http://dapse.unbox.org/`

Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, IWSM-MENSURA '11, pages 116–125, Washington, DC, USA, 2011. IEEE Computer Society.

[4] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proceedings of the 35th International Conference on Software Engineering*, May 2013.

[5] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In M. Lanza, M. D. Penta, and T. Xi, editors, *MSR*, pages 40–49. IEEE, 2012.

[6] R. Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005.

[7] E. Shihab. *An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*. PhD thesis, School of Computing, Queen's University, Kingston, Ontario, Canada, 2012.

[8] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 249–258, Washington, DC, USA, 2010. IEEE Computer Society.

[9] R. Souza and C. Chavez. Characterizing verification of bug fixes in two open source ides. In M. Lanza, M. D. Pent, and T. Xi, editors, *MSR*, pages 70–73. IEEE, 2012.

[10] R. Souza, C. Chavez, and R. Bittencourt. Patterns for cleaning up bug data. In *Proceedings of the First Workshop on Data Analysis Patterns in Software Engineering*. IEEE, May 2013.

[11] R. Souza, C. Chavez, and R. Bittencourt. Patterns for extracting high level information from bug reports. In *Proceedings of the First Workshop on Data Analysis Patterns in Software Engineering*. IEEE, May 2013.

[12] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1074–1083, Piscataway, NJ, USA, 2012. IEEE Press.